PROYECTO - Analizador Sintactico

Curso: Compiladores

De: Ruelas Lope, Rodrigo Alonso
Ciclo: 3
Semestre: V
01 de Mayo del 2024

Contents

1 Introducción

1.1 Justificación

Para los recién aventurados a mundo de la programación, es posible que no estén acostumbrados a los códigos y algoritmos, puede ser un poco complicado al inicio, pero con práctica se puede ir mejorando. Por eso, para darle a los nuevos maestros de la programación, ofrezco un nuevo lenguaje que combina dos lenguajes de programación: PSeint (nivel básico) y C++ (nivel medio), para que puedan ir conociendo el nivel de dificultad que tiene C++, ya que este lenguaje es uno de los más utilizados en nuestros tiempos, pero, para darles un poco de ambientación a este mundo de extensas líneas de código, PSeint se hace presente para darles un apoyo en la dificultad de C++. Este nuevo lenguaje es llamado, PC.

1.2 Objetivo

Crear un nuevo lenguaje de programación (PC), una fusión los lenguajes de programación: PSeint y C++.

1.3 Link del repositorio

https://github.com/RodrigoRuelas/CompiladorPC.git

2 Especificacion Lexica

2.1 Definicion de los tokens

Para las especificaciones léxicas para el lenguaje PC, vamos a dividir en 9 partes, vamos a definirlas a continuación:

1. **Inicio y Fin.** El comienzo y el fin de las líneas de código que leerán nuestro nuevo código. Para ello, son definidas como palabras clave: Inicio y Fin. Tambien, vamos a incluir un comienzo y un fin especial para las funciones.

2. **Atributos.** La designación de elementos tipo número, decimal, lógico y texto que serán reconocidos en el nuevo código. Serán definidos como a palabras clave: int, float, bool, string.

3. **Numeros.** Para los números, como tenemos atributos tipo int (números enteros) y float (números decimales), tenemos que darles una definición. En los números enteros, puede ser cualquier número, ya sea de una o más cifras, pero no de cero cifras. En los números decimales, puede haber números naturales, ceros o nada en la parte entera (12.4; 0.2; 00.54; .43), el punto, que separa la parte entera con la parte decimal es importante para leer los números flotantes. Y tiene que haber una o más cifras en la parte decimal, si no lo negara.

4. **Valores de bool.** Tenemos el atributo tipo bool presente en nuestro código, lo cual, necesita un valor específico. Para ello, vamos a definir sus valores por medio de palabras clave: true y false.

5. **Textos.** Tenemos atributos tipo string, donde su definición es la siguiente: Para los textos, tiene que empezar y terminar con comillas (“ ”) de manera obligatoria, de lo contrario, el código lo negara. Dentro de las comillas, se podrá escribir cualquier texto, donde puede empezar con letras, números o signos, sin obviar el espacio.

6. **Controladores / Palabras Clave.** Tenemos estructuras de control en nuestro código (if, for y while), denotados como: Primero, el bucle condicional (if), que está conformado por palabras clave: Si, Sino, FinSi. Segundo, el bucle repetir (for), conformado por las palabras clave: Para, FinPara. Tercero, el bucle mientras (while), formado por las palabras clave: Mientras, FinMientras. Aparte de las estructuras de control, tenemos otras palabras clave, tales como: Mostrar (para escribir mensajes); Salto (salto de línea para los mensajes); Leer (para leer/dar valor a una variable) y Retornar (para retornar un valor específico de una función).

7. **Funciones y Variables.** Nuestro nuevo lenguaje tiene la disponibilidad de crear IFD o ID, nombres especiales para nombrar una funcion (IFD) o una variable (ID). Estas funciones pueden ser nombradas mediante letras y números, pero con la condición de que el nombre de la función tiene

que empezar y terminar con sub-guiones. Ahora, para nuestras variables, pueden ser nombrados mediante letras y numeros, pero con la condicion de que el nombre de la variables tiene que empezar con una letra, ya sea en mayúscula o minúscula. Los siguientes caracteres de la variable pueden ser números o letras, pero no signos.

8. **Signos y Simbolos.** Tenemos una gran cantidad de signos y símbolos, tales como los paréntesis, la coma, las operaciones (aritméticas, de igualdad), signos de AND y OR, y de incremento y disminucion.

2.2 Expresion Regular

Ahora, vamos a presentar los tokens respectivos con su expresión regular que los representa.

INICIO y FIN	
T_Inicio	'Inicio'
T_Fin	'Fin'
T_Funcion	'Funcion'
T_FinFuncion	'FFuncion'

ATRIBUTOS	
A_int	'int'
A_float	'float'
A_bool	'bool'
A_string	'string'

NUMEROS	
Num	([0-9])+
Num_Dec	([0-9])*'.'([0-9])+

VALORES DE BOOL	
Bool_True	‘True’
Bool_False	‘False’

TEXTOS	
Tex	' " ' ([a-zA-Z]|[0-9])* ' " '

CONTROLADORES / PALABRAS CLAVE	
C_Si	'Si'
C_SiNo	SN'
C_FinSi	FSi'
C_Para	'Para'
C_FinPara	FPara'
C_Mientras	'Mientras'
C_FinMientras	FMientras'
P_Mostrar	'Mostrar'
P_SaltoLinea	'S'
P_Leer	'Leer'
P_Retornar	'Retornar'
P_Romper	'Romper'

FUNCIONES y VARIABLES (ID y IFD)	
ID	[a-zA-Z]([a-zA-Z]|[0-9])*
IFD	' '[a-zA-Z]([a-zA-Z]|[0-9])*' '

SIMBOLOS Y SIGNOS	
S_lpar	'('
S_rpar	')'
S_coma	','
C_And	'&&'
C_Or	'||'
S_sum	'+'
S_res	'-'
S_multi	'*'
S_div	'/'
S_igual	'='
S_mayor	'<'
S_menor	'>'
S_mayorI	'<='
S_menorI	'>='
S_igualdad	'=='
S_desigualdad	'<>'
S_ascender	'::'
S_descender	':'

3 Gramatica LL1

Listing 1: Gramatica LL1

```
S -> T_Inicio E T_Fin
S -> T_Funcion IFD PARAMETRO E T_FinFuncion S
PARAMETRO -> S_lpar PAR S_rpar
PAR -> AT ID PAR2
PAR2 -> S_coma PAR
PAR2 -> ''


E -> VARIABLE
E -> ACCION
E -> MOSTRAR
E -> CONTROL
E -> FUNCION E
E -> LEER
E -> RETORNAR
E -> ROMPER
E -> ''


MOSTRAR -> P_Mostrar MOS1 E
MOS1 -> ID MOS2
MOS1 -> Tex MOS2
MOS1 -> Num MOS2
MOS1 -> FUNCION MOS2
MOS2 -> S_sum MOS1
MOS2 -> ''


VARIABLE -> AT ID EX E
AT -> A_int
AT -> A_float
AT -> A_string
AT -> A_bool
B -> Bool_True
B -> Bool_False

EX -> S_igual VALOR
EX -> ''

VALOR -> S_lpar VALOR S_rpar VALOR1
VALOR -> ID VALOR1
VALOR -> Num VALOR1
```

```
VALOR -> Tex VALOR1
VALOR -> B
VALOR -> FUNCION VALOR1
VALOR1 -> OP VALOR
VALOR1 -> ''

OP -> S_sum
OP -> S_res
OP -> S_multi
OP -> S_div


ACCION -> ID S_igual VALOR E


CONTROL -> SI
CONTROL -> MIENTRAS
CONTROL -> PARA


SI -> C_Si CONDICION E SINO C_FinSi E
SINO -> C_SiNo E
SINO -> ''
CONDICION -> S_lpar EXC S_rpar
EXC -> ID VALOR1 EXC1
EXC -> Num VALOR1 EXC1
EXC -> B EXC1
EXC -> FUNCION VALOR1 EXC1
EXC1 -> OP_L EXC
EXC1 -> ''

MIENTRAS -> C_Mientras CONDICION E C_FinMientras E

PARA -> C_Para S_lpar CONDICIONP S_rpar E C_FinPara E
CONDICIONP -> AT ID S_igual Num S_coma EXC S_coma ID SR

OP_L -> S_mayor
OP_L -> S_menor
OP_L -> S_mayorI
OP_L -> S_menorI
OP_L -> S_igualdad
OP_L -> S_desigualdad
OP_L -> S_And
OP_L -> S_Or

SR -> S_ascender
```

```
SR -> S_descender

FUNCION -> IFD S_lpar EXF S_rpar
EXF -> VALOR EXF1
EXF1 -> S_coma EXF
EXF1 -> ''


LEER -> P_Leer ID E


RETORNAR -> P_Retornar VALOR E


ROMPER -> P_Romper E
```

4 Implementacion en Codigo

Ahora, vamos a implementarlo en codigo, en este caso, en Python.

1. **Librerias** Para, empezar con toda la implementacion, vamos a usar las siguientes librerias:
   a. **import ply.lex as lex** Para construir el analizador lexico de nuestro lenguaje.
   b. **import csv** Para procesar datos de la tabla csv.
   c. **from collections import deque** Para la construccion del arbol sintactico.
   d. **import re** Para poder hallar los terminales y no determinales (division de la regla de izquierda y derecha)

Listing 2: Ejemplo: Import's

```
import ply.lex as lex
import csv
from collections import deque
import re
```

2. **Analizador Lexico** Este analizador ya lo teniamos hecho, solo le hemos hecho unas modificaciones: Uno, en lugar de que imprima type, value, line, position en la consola, seran guardadas en un txt, llamado "Detalles". Dos, la cadena de tokens sera guardada en un archivo txt, llamado "input". Todo esto, lo generara el codigo.

Listing 3: Analizador Lexico

```
# ANALIZADOR LEXICO
# Lista de Tokens
tokens = (
    # Inicio y Fin
    'T_Inicio',
    'T_Fin',
    'T_Funcion',
    'T_FinFuncion',
    # Atributos
    'A_int',
    'A_float',
    'A_bool',
    'A_string',
    # Numeros
    'Num',
    # Valores de Bool
```

```
    'Bool_True',
    'Bool_False',
    # Controladores / Palabras Clave
    'C_Si',
    'C_SiNo',
    'C_FinSi',
    'C_Para',
    'C_FinPara',
    'C_Mientras',
    'C_FinMientras',
    'P_Mostrar',
    'C_SaltoLinea',
    'P_Leer',
    'P_Retornar',
    'P_Romper',
    # Simbolos y Signos
    'S_lpar',
    'S_rpar',
    'S_And',
    'S_Or',
    'S_coma',
    'S_sum',
    'S_res',
    'S_multi',
    'S_div',
    'S_igual',
    'S_mayor',
    'S_menor',
    'S_mayorI',
    'S_menorI',
    'S_igualdad',
    'S_desigualdad',
    'S_ascender',
    'S_descender',
    # Textos
    'Tex',
    # Variables y Funciones (ID)
    'ID',
    'IFD',
)

# Expresiones regulares
# Tokens prioritarios
def t_T_Inicio(t):
  r'Inicio'
  return t
```

```
def t_T_Fin(t):
  r'Fin'
  return t

def t_T_Funcion(t):
  r'Funcion'
  return t

def t_T_FinFuncion(t):
  r'FFuncion'
  return t

def t_A_int(t):
  r'int'
  return t

def t_A_float(t):
  r'float'
  return t

def t_A_bool(t):
  r'bool'
  return t

def t_Bool_True(t):
  r'True'
  return t

def t_Bool_False(t):
  r'False'
  return t

def t_A_string(t):
  r'string'
  return t

def t_C_Si(t):
  r'Si'
  return t

def t_C_SiNo(t):
  r'SN'
  return t

def t_C_FinSi(t):
```

```
    r'FSi'
    return t

def t_C_Para(t):
    r'Para'
    return t

def t_C_FinPara(t):
    r'FPara'
    return t

def t_C_Mientras(t):
    r'Mientras'
    return t

def t_C_FinMientras(t):
    r'FMientras'
    return t

def t_P_Mostrar(t):
    r'Mostrar'
    return t

def t_C_SaltoLinea(t):
    r'S'
    return t

def t_P_Leer(t):
    r'Leer'
    return t

def t_P_Retornar(t):
    r'Retornar'
    return t

def t_P_Romper(t):
    r'Romper'
    return t

# Textos
def t_Tex(t):
    r'("[a-zA-Z0-9- ]*")'
    return t

# Variables y Funciones
def t_ID(t):
```

```
    r'[a-zA-Z][a-zA-Z0-9]*'
    return t
def t_IFD(t):
    r'_[a-zA-Z][a-zA-Z0-9]*_'
    return t

# Numeros
def t_Num(t):
  r'\d+(\.\d+)?'
  if '.' in t.value:
      t.value = float(t.value)
  else:
      t.value = int(t.value)
  return t

def t_S_ascender(t):
  r'::'
  return t

def t_S_descender(t):
  r':'
  return t

# Tokens con expresiones regulares m s simples
t_ignore = '\t'

# Simbolos y Signos
t_S_lpar = r'\('
t_S_rpar = r'\)'
t_S_And = r'&&'
t_S_Or = r'\|\|'
t_S_coma = r','
t_S_sum = r'\+'
t_S_res = r'-'
t_S_multi = r'\*'
t_S_div = r'/'
t_S_igual = r'='
t_S_mayor = r'<'
t_S_menor = r'>'
t_S_mayorI = r'<='
t_S_menorI = r'>='
t_S_igualdad = r'=='
t_S_desigualdad = r'<>'

# Define a rule so we can track line numbers
def t_newline(t):
```

```
    r'\n+'
    t.lexer.lineno += len(t.value)

# Error handling rule
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Leer el contenido del archivo
with open('Codigo Base - Examen Parcial.txt', 'r') as file:
  data = file.read()

# Darle la entrada al lexer
lexer.input(data)

# Lista para almacenar los tokens como diccionarios
tokens_list = []

# Tokenize
while True:
  tok = lexer.token()
  if not tok:
    break  # No hay m s entrada
  # Crear un diccionario para cada token
  token_info = {
    'type': tok.type,
    'value': tok.value,
    'line': tok.lineno,
    'position': tok.lexpos
  }
  tokens_list.append(token_info)
  with open("Input.txt", "w") as f:
   for token in tokens_list:
      f.write(f"{token['type']} ")
   f.write("$\n")  # A adir $ al final despu s de todos los tokens

# Escribir los detalles completos de cada token en token_details.txt
with open("Detalles.txt", "w") as f:
  for token in tokens_list:
      f.write(f"Type: {token['type']}, Value: {token['value']}, Line: {token['li
```

1. **Analizador sintactico - Funciones** Para empezar a modelar nuestro analizador sintactico, vamos a requerir de unas funciones para desarrollar

la tabla sintactica.

Listing 4: Funcion: Identificar terminal y no terminal

```
def identificar_terminal_noterminal(rules):
    non_terminals = set()
    all_symbols = set()

    for rule in rules:
        lhs, rhs = re.split(r'\s*->\s*', rule.strip())
        non_terminals.add(lhs)
        all_symbols.update(rhs.split())

    return non_terminals, all_symbols - non_terminals
```

Listing 5: Funcion: Leer Gramatica LL1

```
def leer_gramatica(filename):
  with open(filename, 'r') as file:
    rules = [line.strip() for line in file if line.strip()]
  return rules
```

2. Analizador sintactico - First Para comenzar a desarrollar la tabla sintactica, requerimos de los conjuntos de primeros de la gramatica.

Listing 6: Funcion: Hallar conjuntos de primeros

```
# Para hallar los conjuntos de primeros
def hallar_primeros(rules):
    firsts = {}
    change = True

    # Inicializar FIRST para cada no terminal y terminal
    non_terminals, terminals = identificar_terminal_noterminal(rules)
    for symbol in non_terminals.union(terminals):
        firsts[symbol] = set()
        if symbol in terminals:
            firsts[symbol].add(symbol)

    while change:
        change = False
        for rule in rules:
            lhs, rhs = rule.split('->')
            lhs = lhs.strip()
            rhs = rhs.strip().split()

            original_first = firsts[lhs].copy()
            can_be_empty = True
```

```
            for symbol in rhs:
                if not can_be_empty:
                    break

                firsts[lhs].update(firsts[symbol] - {EPSILON})

                if EPSILON not in firsts[symbol]:
                    can_be_empty = False

            if can_be_empty:
                firsts[lhs].add(EPSILON)

            if firsts[lhs] != original_first:
                change = True

        return firsts
```

2. **Analizador sintactico - Follow** El siguiente paso es hallar los conjuntos de siguientes. Para ello, vamos a emplear el siguiente codigo.

Listing 7: Funcion: Hallar conjuntos de siguientes

```
# Para hallar los conjuntos de siguientes
def hallar_siguientes(rules, firsts, start_symbol):
 follows = {non_terminal: set() for non_terminal in firsts}
 follows[start_symbol].add('$')  # A adir el s mbolo de fin de archivo al

 changed = True
 while changed:
    changed = False
    for rule in rules:
        parts = rule.split('->')
        if len(parts) < 2:
            continue
        lhs = parts[0].strip()
        rhs = parts[1].strip().split()

        trailer = set(follows[lhs])
        for i in reversed(range(len(rhs))):
            symbol = rhs[i]
            if symbol in follows:  # Solo considerar no terminales
                before_update = len(follows[symbol])
                # A adir trailer, excluyendo expl citamente   psilon   si  e
                follows[symbol].update(trailer)
                if len(follows[symbol]) > before_update:
                    changed = True
```

```
                    # Si el s mbolo puede derivar en  psilon , actualizar el tr
                    if ' ' in firsts.get(symbol, set()):
                        trailer.update(x for x in firsts[symbol] if x != ' ')
                    else:
                        trailer = set(firsts.get(symbol, set()))
                else:
                    if symbol != ' ':
                        trailer = {symbol}

    # Eliminar cualquier conjunto FOLLOW vac o
    non_empty_follows = {}
    for non_terminal, follow_set in follows.items():
        if follow_set:
            non_empty_follows[non_terminal] = follow_set

    return non_empty_follows
```

3. **Analizador sintactico - Tabla** Como ultimo paso, tenemos que elaborar la grafica. Para ello, vamos a eleborar los siguientes codigos.

Listing 8: Funcion: Elaborar Tabla

```
# Para hallar la tabla de an lisis sint ctico
def leer_conjuntos(filename):
    sets = {}
    with open(filename, 'r') as file:
        for line in file:
            line = line.strip()
            if line:
                non_terminal, elements = line.split(':')
                elements = elements.split(',')
                sets[non_terminal.strip()] = set(e.strip() if e.strip() != "''"
    return sets

def leer_gramatica_tabla(filename):
    rules = {}
    with open(filename, 'r') as file:
        current_nt = None
        for line in file:
            line = line.strip()
            if line:
                if '->' in line:
                    lhs, rhs = line.split('->')
                    lhs, rhs = lhs.strip(), rhs.strip()
                    if lhs not in rules:
                        rules[lhs] = []
                    current_nt = lhs
```

```
                    rules[lhs].append(rhs.split())
                elif current_nt:
                    # Continuation of the previous non-terminal's productions
                    rules[current_nt].append(line.split())
    return rules

def construir_tabla(rules, firsts, follows):
    terminals = set(term for terms in firsts.values() for term in terms if term
    non_terminals = set(rules.keys())
    table = {nt: {t: [] for t in terminals} for nt in non_terminals}

    for nt, productions in rules.items():
        for production in productions:
            first = calcular_primeros(production, firsts)
            for symbol in first - {''}:
                table[nt][symbol].append(production)
            if '' in first:
                for symbol in follows[nt]:
                    table[nt][symbol].append(production)

    return table

def calcular_primeros(sequence, firsts):
    result = set()
    for symbol in sequence:
        result.update(firsts.get(symbol, {symbol}))
        if '' not in firsts.get(symbol, {}):
            result.discard('')
            break
    else:
        result.add('')
    return result

def imprimir_tabla(table, filename):
    with open(filename, 'w') as file:
        for nt, row in table.items():
            file.write(f"{nt}:\n")
            for terminal, productions in row.items():
                if productions:
                    production_str = ' | '.join(' '.join(prod) for prod in pro
                    file.write(f"  {terminal}: {production_str}\n")
```

4. **Analizador sintactico - Funciones y main** Para completar todo el analisis de la gramatica LL1, vamos a tener en cuenta los siguientes codigos: funcion de imprimir conjuntos y llamado de funciones (main1).

Listing 9: Funcion: Funcion de imprimir conjuntos y llamado de funciones (main1)

```
# Para escribir txt con los conjuntos de primeros y siguientes
def escribir_conjuntos(sets, filename, set_type):
    with open(filename, 'w') as file:
    for symbol in sorted(sets):
        set_formatted = ', '.join(sorted(sets[symbol]))
        file.write(f"{symbol}: {set_formatted}\n")

    print(f"Los conjuntos de {set_type} han sido creado y guardado en {filena

EPSILON = "'"  # Aseg rate de que EPSILON coincida con c mo lo representa
terminals = set()

filename = 'LL1 - oficial.txt'
rules = leer_gramatica(filename)
start_symbol = rules[0].split('->')[0].strip()  # Asumir que el primer no te

# Compute FIRST sets
first = hallar_primeros(rules)

# Filtrar cualquier entrada vac a antes de escribir a archivo
if '' in first:
    del first['']

# Escribir los conjuntos FIRST y FOLLOW a archivos
escribir_conjuntos(first, 'first.txt', 'FIRST')

# Compute FOLLOW sets using the computed FIRST sets
filename_firsts = read_firsts('first.txt')
follows = hallar_siguientes(rules, filename_firsts, start_symbol)
escribir_conjuntos(follows, 'follow.txt', 'FOLLOW')

firsts_filename = 'first.txt'
follows_filename = 'follow.txt'

rules_T = leer_gramatica_tabla(filename)
firsts_T = leer_conjuntos(firsts_filename)
follows_T = leer_conjuntos(follows_filename)

ll1_table = construir_tabla(rules_T, firsts_T, follows_T)
output_filename = 'll1_table.txt'
imprimir_tabla(ll1_table, output_filename)
print(f"LL(1) tiene su tabla, nombrada {output_filename}")
```

8. **Analizador sintactico - Analisis de input** Para completar todo el fun-

cionamiento del analizador sintactico, tenemos que usar nuestra tabla, que en este caso, lo estamos ingresando nosotros, no estamos usando la tabla que genera nuestro codigo - parte 1. Tambien, estamos usando un input, que es generado por el analizador sintactico. Para implementar el analisis, vamos usar los siguientes codigos, que tienen las siguientes caracteristicas: cargar la tabla de formato csv, dos analizadores del input (uno, para mostrar los pasos de la pila y el otro para dar forma al arbol), para crear el arbol y ser exportado a un archivo .dot (para Graphviz) y para obtener el index del input. Ademas, tenemos una clase para generar los nodos y los nodos-hijos para el arbol.

Listing 10: Funcion: Analisis (main2)

```
# ANALISIS

class Nodo:
    def __init__(self, simbolo):
        self.simbolo = simbolo
        self.hijos = []

    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)

def cargar_tabla(archivo):
  tabla = {}
  with open(archivo, newline='') as csvfile:
      reader = csv.reader(csvfile)
      for fila in reader:
          estado = fila[0]
          transiciones = fila[1:]
          tabla[estado] = transiciones
  return tabla

def analizar_entrada_P(entrada, tabla):
  pila = ['$', 'S']  # Pila inicial con $ en el fondo y el s mbolo inicial
  idx_entrada = 0
  while len(pila) > 0:
      print("Pila:", pila)
      print("Entrada:", entrada[idx_entrada:])
      print("")

      simbolo_actual = pila.pop()
      if idx_entrada < len(entrada):
          simbolo_entrada = entrada[idx_entrada]

          if simbolo_actual in tabla:
              transiciones = tabla[simbolo_actual]
```

```
                accion = transiciones[get_index(simbolo_entrada)]

                if accion != '-':
                    if accion != 'e':
                        produccion = accion.split()
                        pila.extend(reversed(produccion))
                    else:
                        continue
                else:
                    continue
            else:
                if simbolo_actual == simbolo_entrada:
                    idx_entrada += 1
                else:
                    return False
        else:
            return False

    return idx_entrada == len(entrada) and not pila

def analizar_entrada_A(entrada, tabla):
    raiz = Nodo('S')  # Nodo raiz del arbol
    pila = [('$', None), ('S', raiz)]  # Se agrega el nodo raiz a la pila

    idx_entrada = 0
    while len(pila) > 0:
        simbolo_actual, nodo_actual = pila.pop()

        if idx_entrada < len(entrada):
            simbolo_entrada = entrada[idx_entrada]
        else:
            break  # Salida anticipada si no hay mas entrada

        if simbolo_actual in tabla:
            transiciones = tabla[simbolo_actual]
            accion = transiciones[get_index(simbolo_entrada)]

            if accion != '-':
                if accion == 'e':  # Manejo de produccion epsilon
                    nodo_epsilon = Nodo('e')
                    nodo_actual.agregar_hijo(nodo_epsilon)
                else:
                    produccion = accion.split()
                    # Añadir nodos a la pila en orden inverso para procesar el
                    for simbolo in reversed(produccion):
                        nuevo_nodo = Nodo(simbolo)
```

```
                    nodo_actual.agregar_hijo(nuevo_nodo)
                    pila.append((simbolo, nuevo_nodo))
        else:
            # Si es un s mbolo terminal, verificar coincidencia
            if simbolo_actual == simbolo_entrada:
                idx_entrada += 1
            else:
                return False, None  # Retorno inmediato si no hay coincidencia

    return idx_entrada == len(entrada) and not pila, raiz
# Asegurar que se consumi  toda la entrada y la pila est  vac a


def generar_dot(nodo, buffer, id=0, parent_id=None):
    # Si el nodo es None (nodo vac o para epsilon), no hacer nada
    if nodo is None:
        return id

    # Definir el nombre del nodo usando su ID para garantizar unicidad
    node_name = f'node{id}'

    # Crear la declaraci n del nodo para Graphviz
    buffer.append(f'{node_name} [label="{nodo.simbolo}"];')

    # Si el nodo tiene un padre, a adir una arista de parent_id a node_name
    if parent_id is not None:
        buffer.append(f'{parent_id} -> {node_name};')

    # Recorrer todos los hijos del nodo actual
    child_id = id + 1
    for hijo in nodo.hijos:
        # Llamar recursivamente para cada hijo
        child_id = generar_dot(hijo, buffer, child_id, node_name)

    # Devolver el pr ximo ID disponible
    return child_id

def exportar_arbol_a_dot(raiz):
    buffer = ['digraph G {']
    generar_dot(raiz, buffer)
    buffer.append('}')
    return '\n'.join(buffer)

def get_index(simbolo):
    indice_simbolos = {
        'T_Inicio': 0,
```

```
        'T_Fin': 1,
        'T_Funcion': 2,
        'IFD': 3,
        'T_FinFuncion': 4,
        'S_lpar': 5,
        'S_rpar': 6,
        'ID': 7,
        'S_coma': 8,
        'P_Mostrar': 9,
        'Tex': 10,
        'Num': 11,
        'S_sum': 12,
        'A_int': 13,
        'A_float': 14,
        'A_string': 15,
        'A_bool': 16,
        'Bool_True': 17,
        'Bool_False': 18,
        'S_igual': 19,
        'S_res': 20,
        'S_multi': 21,
        'S_div': 22,
        'C_Si': 23,
        'C_FinSi': 24,
        'C_SiNo': 25,
        'C_Mientras': 26,
        'C_FinMientras': 27,
        'C_Para': 28,
        'C_FinPara': 29,
        'S_mayor': 30,
        'S_menor': 31,
        'S_mayorI': 32,
        'S_menorI': 33,
        'S_igualdad': 34,
        'S_desigualdad': 35,
        'S_And': 36,
        'S_Or': 37,
        'S_ascender': 38,
        'S_descender': 39,
        'P_Leer': 40,
        'P_Retornar': 41,
        'P_Romper': 42,
        '$': 43
    }
    return indice_simbolos.get(simbolo, -1)
```

```
archivo_tabla = 'LL1 - PS.csv'
tabla = cargar_tabla(archivo_tabla)

archivo_entrada = 'Input.txt'  # Nombre del archivo con la entrada
with open(archivo_entrada, 'r') as file:
    entrada = file.read().split()  # Lee el archivo y divide la entrada en t

if analizar_entrada_P(entrada, tabla):
    print("La entrada es aceptada por la tabla.")
else:
    print("La entrada no es aceptada por la tabla.")

aceptado, raiz = analizar_entrada_A(entrada, tabla)
if aceptado:
    codigo_dot = exportar_arbol_a_dot(raiz)
    with open('arbol_sintactico.dot', 'w') as f:
        f.write(codigo_dot)
```

5 Ejemplos

1. Vamos a ejecutar el codigo elaborado con tres ejemplos input de nuestro lenguaje propuesto, mostrarndo los siguientes resultados: Arbol.

a) CODIGOS EJEMPLOS.

Listing 11: Ejemlo 1

```
Inicio
        int f = 1
        int a

        Mostrar "Ingresar numero"
        Leer a

        Para (int i=1, i<=a, i:)
                f = f * i
        FPara

        Mostrar f
Fin
```

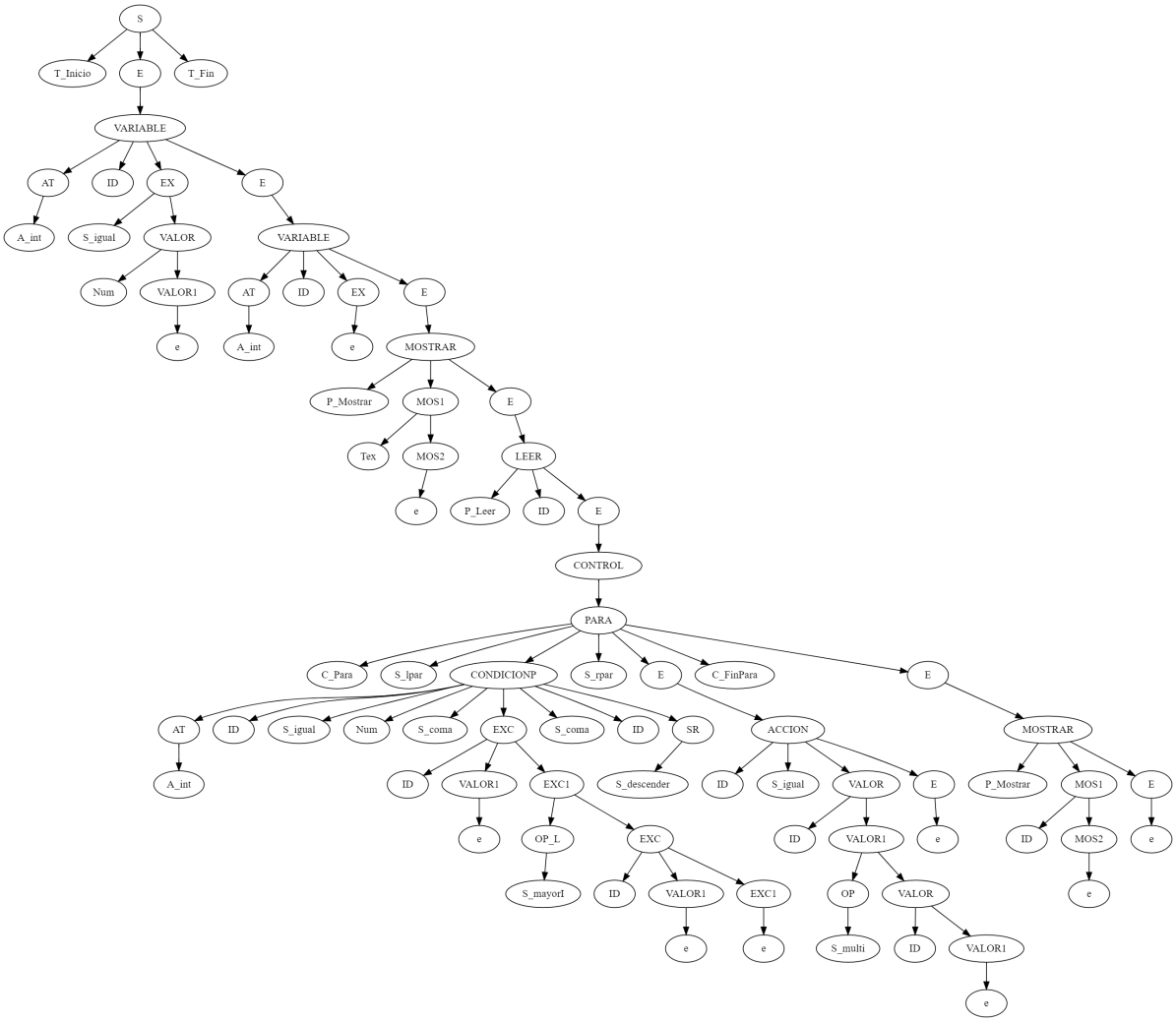
Listing 12: Ejemlo 2

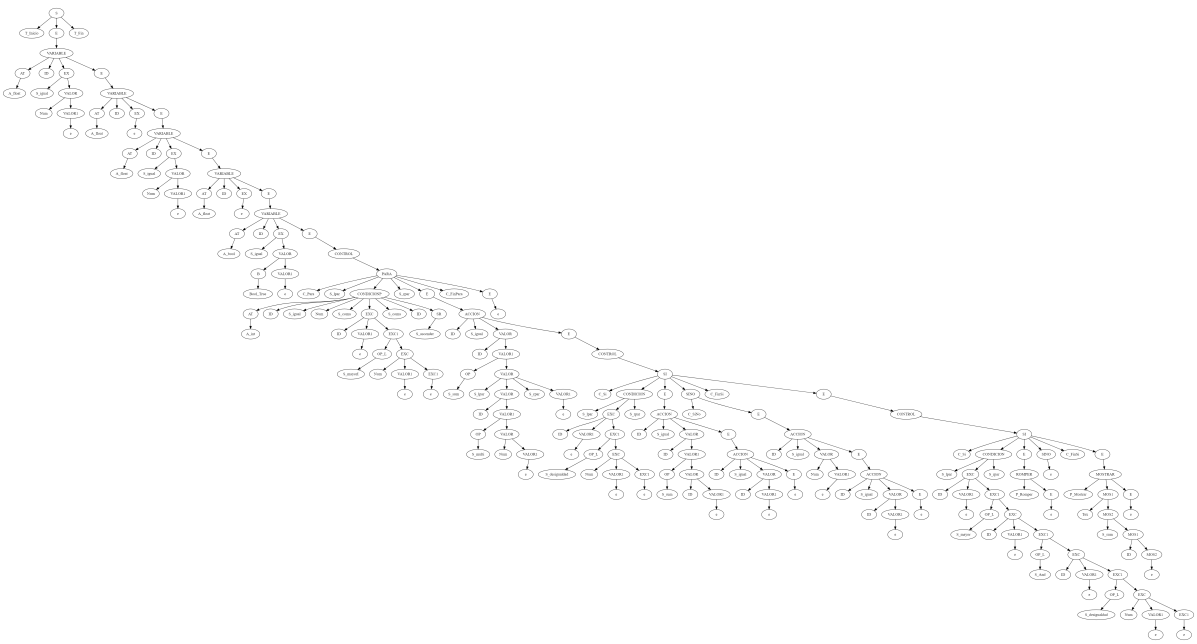
```
Inicio
        float limit = 10.1
        float aprox
        float aproxA = 0
        float Total
  bool Valor = True
        Para(int i=0, i<=15, i::)
                aprox = aprox + (i * 1.5)
                Si (aproxA <> 0)
                        Total = aprox + aproxA
                        aproxA = aprox
                SN
                        Total = 0
                        aproxA = aprox
                FSi

                Si (Total < limit && limit <> 0)
                        Romper
                FSi
                Mostrar "Total" + Total
        FPara
Fin
```

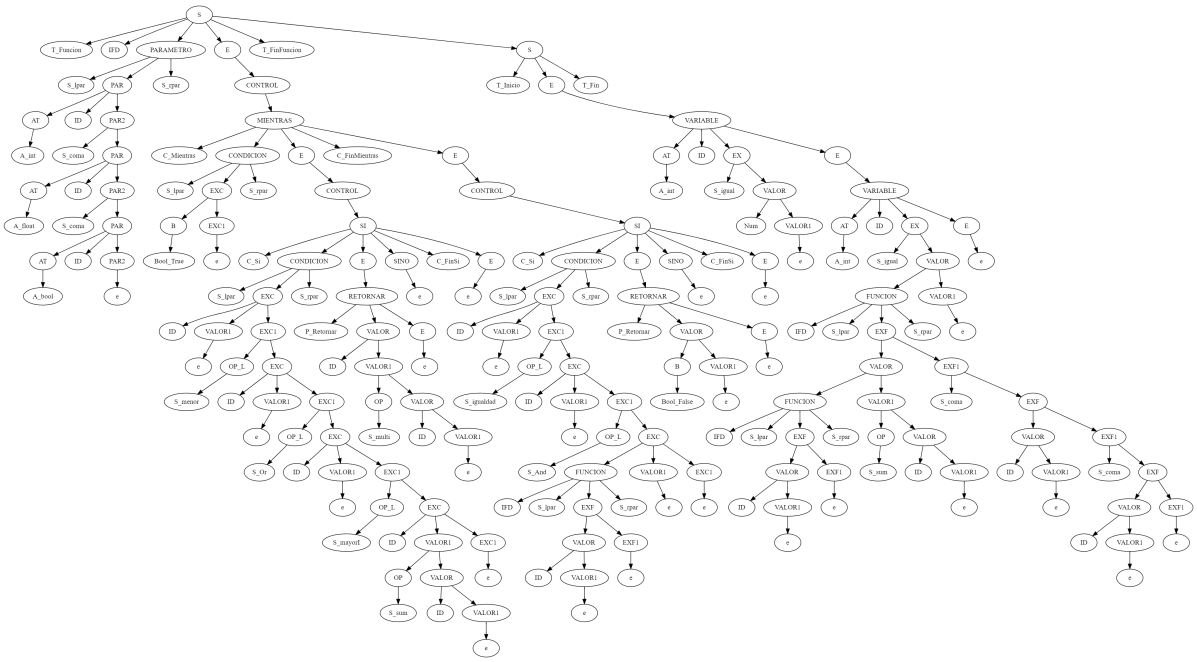
Listing 13: Ejemlo 3

```
Funcion _f_(int x, float y, bool z)
        Mientras(True)
                Si(x>y || y<=x+y)
                        Retornar z*x
                FSi
        FMientras
        Si(x==y && _f_(x))
                Retornar False
        FSi
FFuncion

Inicio
        int x=10
        int y=_f_(_f_(x)+x,x,x)
Fin
```

b) ARBOL.


S
T_Inicio
E
T_Fin
VARIABLE
AT
ID
EX
E
A_int
S_igual
VALOR
VARIABLE
Num
VALOR1
AT
ID
EX
E
e
A_int
e
MOSTRAR
P_Mostrar
MOS1
E
Tex
MOS2
LEER
e
P_Leer
ID
E
CONTROL
PARA
C_Para
S_lpar
CONDICIONP
S_rpar
E
C_FinPara
E
AT
ID
S_igual
Num
S_coma
EXC
S_coma
ID
SR
ACCION
MOSTRAR
A_int
ID
VALOR1
EXC1
S_descender
ID
S_igual
VALOR
E
P_Mostrar
MOS1
E
e
OP_L
EXC
ID
VALOR1
e
ID
VALOR1
e
ID
MOS2
e
S_mayorI
ID
VALOR1
EXC1
OP
VALOR
e
e
S_multi
ID
VALOR1
e





6 Analizador Semantico

1. Tabla de simbolos

Con la ayuda de la tabla de simbolos, vamos a identificar todos los tipos de valores que se encuentran en nuestro codigo.

NOTA. No hay codigo de ejecucion.

2. Verificacion de tipos

En este paso, vamos a verificar si la expresion ingresada esta de acuerdo a nuestras reglas de inferencia establecidas.

Listing 14: Reglas de inferencia

```
int OP int = int
int OP float = float
float OP int = float
float OP float = float
int OP–L int = int
int OP–L float = float
float OP–L int = float
float OP–L float = float
```

7 Conclusión

Lo que puedo concluir de todo este proyecto, es que la primera parte del proyecto fue de mi agrado, la construccion de un lenguaje y darle forma con la ayuda de los tokens y la gramatica LL1, fue de lo mejor. Aun hay partes que podemos mejorar en la parte lexica y sintactica, pero fueron me gustaron esa parte.

La segunda mitad, empezo el problema, con la implementacion de la tabla de simboloes y la verificacion, que tristemente me queda atrancado. Pero, fueron partes interesantes, cuando se hacia a mano dichas funciones. Y la ultima parte, generacion de codigo, fue muy interesante, cuando se hacia a mano y a computadora (programa online), tristemete, no puede implementarlo en codigo.

Para terminar, este proyecto fue muy interesante al comienzo pero la segunda mitad fue complicado seguir la corriente, pero sus conceptos fueron interesantes.